

Introduction:

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- * ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- * Featurization: feature extraction, transformation, dimensionality reduction, and selection
- * Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- * Persistence: saving and load algorithms, models, and Pipelines
- * Utilities: linear algebra, statistics, data handling, etc.

Objective:

1. Spark Classification Task

Use one of the following datasets

- **Absenteeism at work:**
<https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work>
- **Immunotherapy Dataset:**
<https://archive.ics.uci.edu/ml/datasets/Immunotherapy+Dataset#>

Perform the following tasks

- Use the following Classification Algorithms: Naïve Bayes, Decision Tree and Random Forest for the same attribute classification**
- Report the Confusion matrix, Accuracy based on FMeasure, Precision & Recall for all the algorithms**
- State the reasons on why one of algorithms out performs the rest.**

Procedure:

1. Read the data set and convert the column data from string to float/double type. We perform the classification based on the columns "Month of Absence", "Day of the week", "Height", "Travel expenses", "Distance", "Body Mass Index".

2. We split the data into train-70%, Test-30%. Finally, we evaluate the model and predict the results of the test data set. Then calculate the confusion matrix using the above columns and find the precision and recall values.

1. Decision Tree:

Decision trees and their ensembles are popular methods for the machine learning tasks of classification and regression. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. Tree ensemble algorithms such as random forests and boosting are among the top performers for classification and regression tasks.

"spark.mllib" supports decision trees for binary and multi class classification and for regression, using both continuous and categorical features. The implementation partitions data by rows, allowing distributed training with millions of instances.

The decision tree is a greedy algorithm that performs a recursive binary partitioning of the feature space. The tree predicts the same label for each bottommost (leaf) partition. Each partition is chosen greedily by selecting the best split from a set of possible splits, in order to maximize the information gain at a tree node.

Decision Tree Code:

We create a vector assembler on input data columns "label (Height)" and "Distance" and use the DecisionTreeClassifier on indexed label and indexed features

```
data = spark.read.load("Absenteeism_at_work.csv", format="csv", header=True, delimiter=",")
data = data.withColumn("MOA", data["Month of absence"] - 0).withColumn("label", data["Height"] - 0). \
    withColumn("ROA", data["Reason for absence"] - 0). \
    withColumn("distance", data["Distance from Residence to Work"] - 0). \
    withColumn("BMI", data["Body mass index"] - 0)
#data.show()

assem = VectorAssembler(inputCols=["label", "distance"], outputCol='features')
data = assem.transform(data)

## Index labels, adding metadata to the label column.
## Fit on whole dataset to include all labels in index.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
## Automatically identify categorical features, and index them.
## We specify maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer = \
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Train a DecisionTree model.
dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures")

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])

# Train model. This also runs the indexers.
model = pipeline.fit(trainingData)
```

Decision Tree Confusion Matrix:

The screenshot shows the PyCharm IDE with a project named 'sparkdemo'. The file explorer on the left shows a directory structure with files like 'absenteeism_at_work.csv', 'adult.csv', 'adult_new.txt', 'DecisionTree.py', 'derby.log', 'diag.txt', 'diagnosis.txt', 'immunotherapy.csv', and 'immunotherapy.txt'. The 'DecisionTree.py' file is open in the editor, showing the following code:

```
1 import ...
11 os.environ["SPARK_HOME"] = "C:\\Spark\\spark-2.3.1-bin-hadoop2.7"
12 os.environ["HADOOP_HOME"] = "C:\\winutils"
13
14 from pyspark.python.pyspark.shell import spark
15
16 data = spark.read.load("Absenteeism_at_work.csv", format="csv", header=True, delimiter=",")
17 data = data.withColumn("MOA", data["Month of absence"] - 0).withColumn("label", data["Height"] - 0). \
18     withColumn("ROA", data["Reason for absence"] - 0). \
19     withColumn("distance", data["Distance from Residence to Work"] - 0). \
20     withColumn("BMI", data["Body mass index"] - 0)
21 #data.show()
22
23 assem = VectorAssembler(inputCols=["label", "distance"], outputCol='features')
```

The Run console shows the following output:

```
Using Python version 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017 18:11:49)
SparkSession available as 'spark'.
2018-07-27 15:51:39 WARN ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
2018-07-27 15:51:44 WARN Utils:66 - Truncated the string representation of a plan since it was too large. This behavior can be adjusted by setting 'spark.debug.maxToStringFields' in
+-----+
|prediction|indexedLabel| features|
+-----+
| 1.0| 1.0|[172.0,11.0]|
| 1.0| 1.0|[172.0,11.0]|
| 1.0| 1.0|[172.0,11.0]|
| 1.0| 1.0|[172.0,11.0]|
| 1.0| 1.0|[172.0,11.0]|
| 1.0| 1.0|[172.0,11.0]|
+-----+
only showing top 5 rows

DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4e739fde6533edb8a359) of depth 5 with 21 nodes
```

Decision Tree Results:

The screenshot shows the PyCharm IDE with the same project and file structure as the previous image. The 'DecisionTree.py' file is open, showing the same code as before. The Run console shows the following output:

```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4e739fde6533edb8a359) of depth 5 with 21 nodes
Decision Tree - Test Accuracy = 0.967742
Decision Tree - Test Error = 0.0322581
The Confusion Matrix for Decision Tree Model is :
[[0 0 ... 0 0]
 [0 0 ... 0 0]
 [0 0 ... 0 0]
 ...
 [2 0 ... 0 0]
 [0 0 ... 0 0]
 [5 0 ... 0 0]]
The precision score for Decision Tree Model is: 0.02972972972972973
The recall score for Decision Tree Model is: 0.02972972972972973

Process finished with exit code 0
```

1. Naive Bayes:

Naive Bayes is a simple multiclass classification algorithm with the assumption of independence between every pair of features. Naive Bayes can be trained very efficiently. Within a single pass to the training data, it computes the conditional probability distribution of each feature given label, and then it applies Bayes' theorem to compute the conditional probability distribution of label given an observation and use it for prediction.

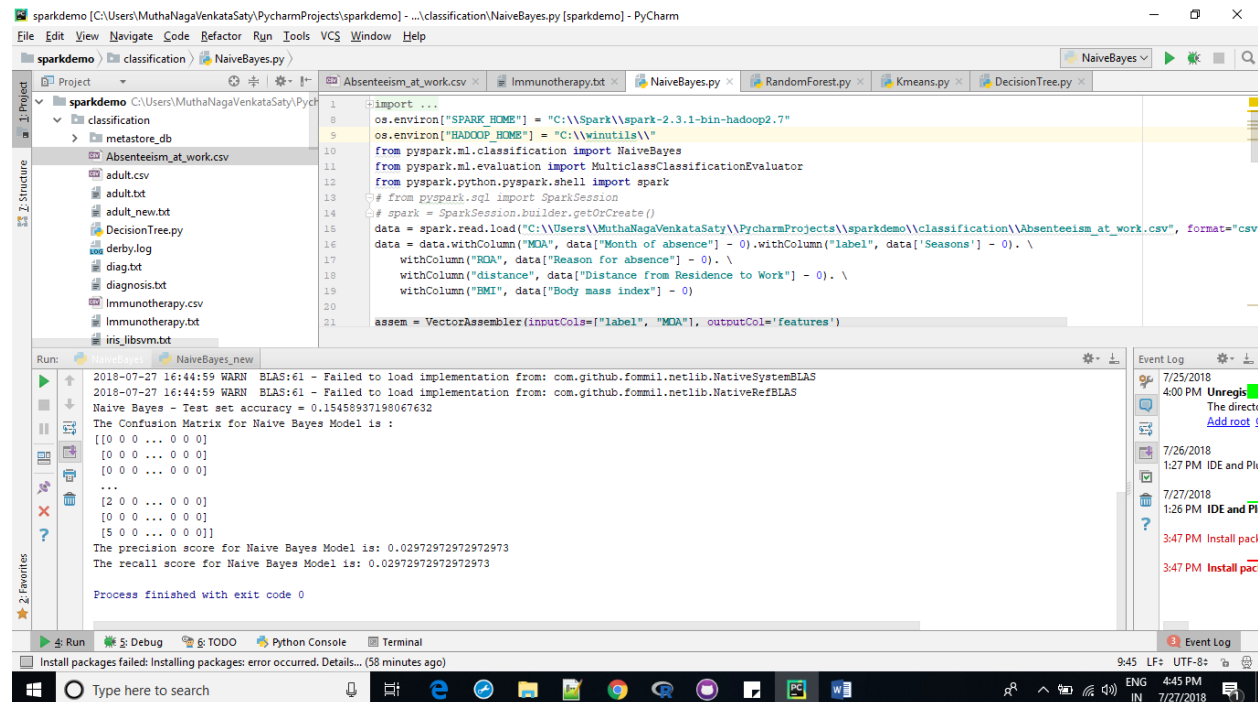
"spark.mllib" supports multinomial naive Bayes and Bernoulli naive Bayes. These models are typically used for document classification. Within that context, each observation is a document and each feature represents a term whose value is the frequency of the term (in multinomial naive Bayes) or a zero or one indicating whether the term was found in the document (in Bernoulli naive Bayes). Feature values must be nonnegative. The model type is selected with an optional parameter "multinomial" or "bernoulli" with "multinomial" as the default. Additive smoothing can be used by setting the parameter λ (default to 1.0). For document classification, the input feature vectors are usually sparse, and sparse vectors should be supplied as input to take advantage of sparsity. Since the training data is only used once, it is not necessary to cache it.

Naive Bayes Code:

NaiveBayes implements multinomial naive Bayes. It takes an RDD of LabeledPoint and an optionally smoothing parameter lambda as input, and output a NaiveBayesModel, which can be used for evaluation and prediction.

```
12 from pyspark.python.pyspark.shell import spark
13 # from pyspark.sql import SparkSession
14 # spark = SparkSession.builder.getOrCreate()
15 data = spark.read.load("C:\\Users\\MuthaNagaVenkataSaty\\PycharmProjects\\sparkdemo\\classification\\Absenteeism_at_work.csv", format="csv",
16                       options={"header": "true", "inferSchema": "true"})
17 data = data.withColumn("MDA", data["Month of absence"] - 0).withColumn("label", data["Seasons"] - 0). \
18             withColumn("ROA", data["Reason for absence"] - 0). \
19             withColumn("distance", data["Distance from Residence to Work"] - 0). \
20             withColumn("BMI", data["Body mass index"] - 0)
21 assem = VectorAssembler(inputCols=["label", "MDA"], outputCol='features')
22 data = assem.transform(data)
23 # Split the data into train and test
24 splits = data.randomSplit([0.7, 0.3], 1000)
25 train = splits[0]
26 test = splits[1]
27 # create the trainer and set its parameters
28 nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
29 # train the model
30 model = nb.fit(train)
31 # select example rows to display.
32 predictions = model.transform(test)
33 # compute accuracy on the test set
34 evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction",
35                                              metricName="accuracy")
36 y_true = data.select("BMI").rdd.flatMap(lambda x: x).collect()
37 y_pred = data.select("ROA").rdd.flatMap(lambda x: x).collect()
38 accuracy = evaluator.evaluate(predictions)
39 confusionmatrix = confusion_matrix(y_true, y_pred)
40 precision = precision_score(y_true, y_pred, average='micro')
41 recall = recall_score(y_true, y_pred, average='micro')
```

Naive Bayes Results:



The screenshot shows the PyCharm IDE with a project named 'sparkdemo'. The file explorer on the left shows a directory structure with files like 'absenteeism_at_work.csv', 'adult.csv', 'adult.txt', 'adult_new.txt', 'DecisionTree.py', 'derby.log', 'diag.txt', 'diagnosis.txt', 'immunotherapy.csv', 'immunotherapy.txt', and 'iris.libsvm.txt'. The main editor window displays the 'NaiveBayes.py' file, which contains the following code:

```
1 import ...
2 os.environ["SPARK_HOME"] = "C:\\Spark\\spark-2.3.1-bin-hadoop2.7"
3 os.environ["HADOOP_HOME"] = "C:\\winutils\\"
4 from pyspark.ml.classification import NaiveBayes
5 from pyspark.ml.evaluation import MulticlassClassificationEvaluator
6 from pyspark.python.pyspark.shell import spark
7 # from pyspark.sql import SparkSession
8 # spark = SparkSession.builder.getOrCreate()
9 data = spark.read.load("C:\\Users\\MuthaNagaVenkataSaty\\PycharmProjects\\sparkdemo\\classification\\absenteeism_at_work.csv", format="csv")
10 data = data.withColumn("MOA", data["Month of absence"] - 0).withColumn("label", data["Seasons"] - 0). \
11     withColumn("ROA", data["Reason for absence"] - 0). \
12     withColumn("distance", data["Distance from Residence to Work"] - 0). \
13     withColumn("BMI", data["Body mass index"] - 0)
14 assem = VectorAssembler(inputCols=["label", "MOA"], outputCol="features")
```

The Run console at the bottom shows the following output:

```
2018-07-27 16:44:59 WARN BLAS:61 - Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
2018-07-27 16:44:59 WARN BLAS:61 - Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
Naive Bayes - Test set accuracy = 0.15458937198067632
The Confusion Matrix for Naive Bayes Model is :
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 ...
 [2 0 0 0 0 0]
 [0 0 0 0 0 0]
 [5 0 0 0 0 0]]
The precision score for Naive Bayes Model is: 0.02972972972972973
The recall score for Naive Bayes Model is: 0.02972972972972973
Process finished with exit code 0
```

3. Random Forest:

Random forests are ensembles of decision trees. Random forests are one of the most successful machine learning models for classification and regression. They combine many decision trees in order to reduce the risk of overfitting. Like decision trees, random forests handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions.

spark.mllib supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features. spark.mllib implements random forests using the existing decision tree implementation. Please see the decision tree guide for more information on trees.

Random Forest Code:

The first two parameters we mention are the most important, and tuning them can often improve performance:

numTrees: Number of trees in the forest.

Increasing the number of trees will decrease the variance in predictions, improving the model's test-time accuracy.

Training time increases roughly linearly in the number of trees.

maxDepth: Maximum depth of each tree in the forest.

Increasing the depth makes the model more expressive and powerful. However, deep trees take longer to train and are also more prone to overfitting.

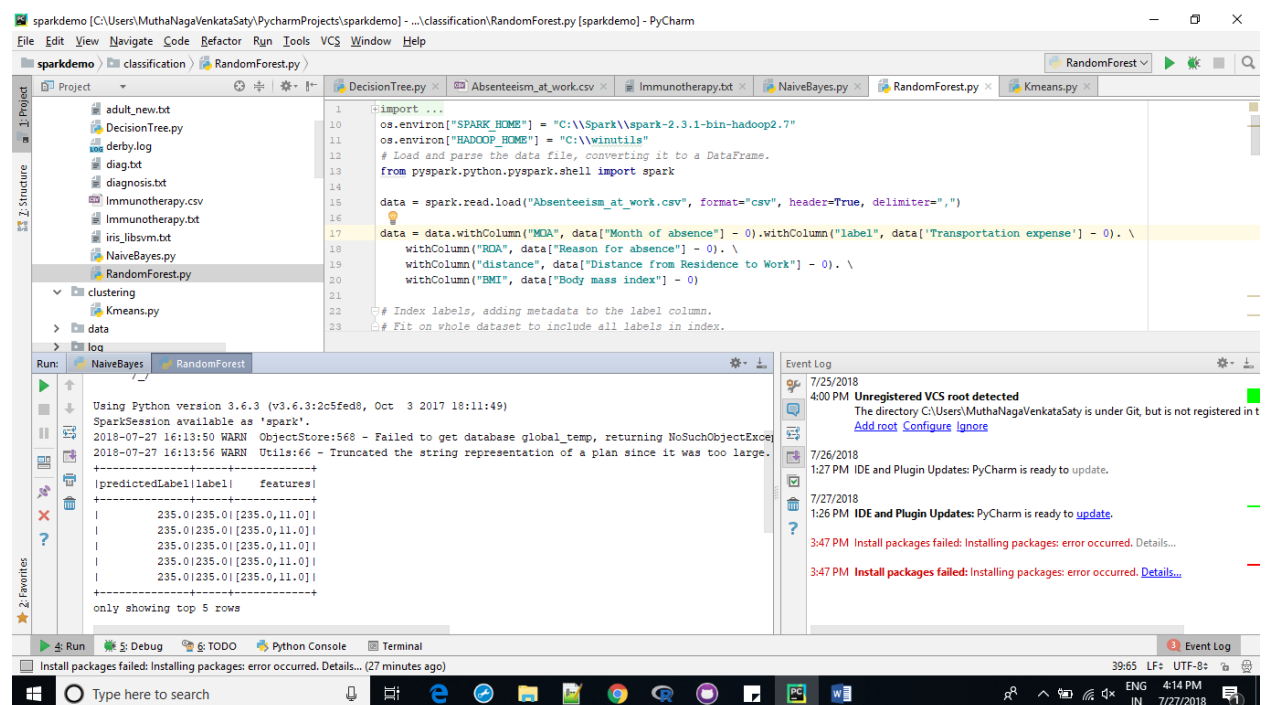
In general, it is acceptable to train deeper trees when using random forests than when using a single decision tree. One tree is more likely to overfit than a random forest (because of the variance reduction from averaging multiple trees in the forest).

```

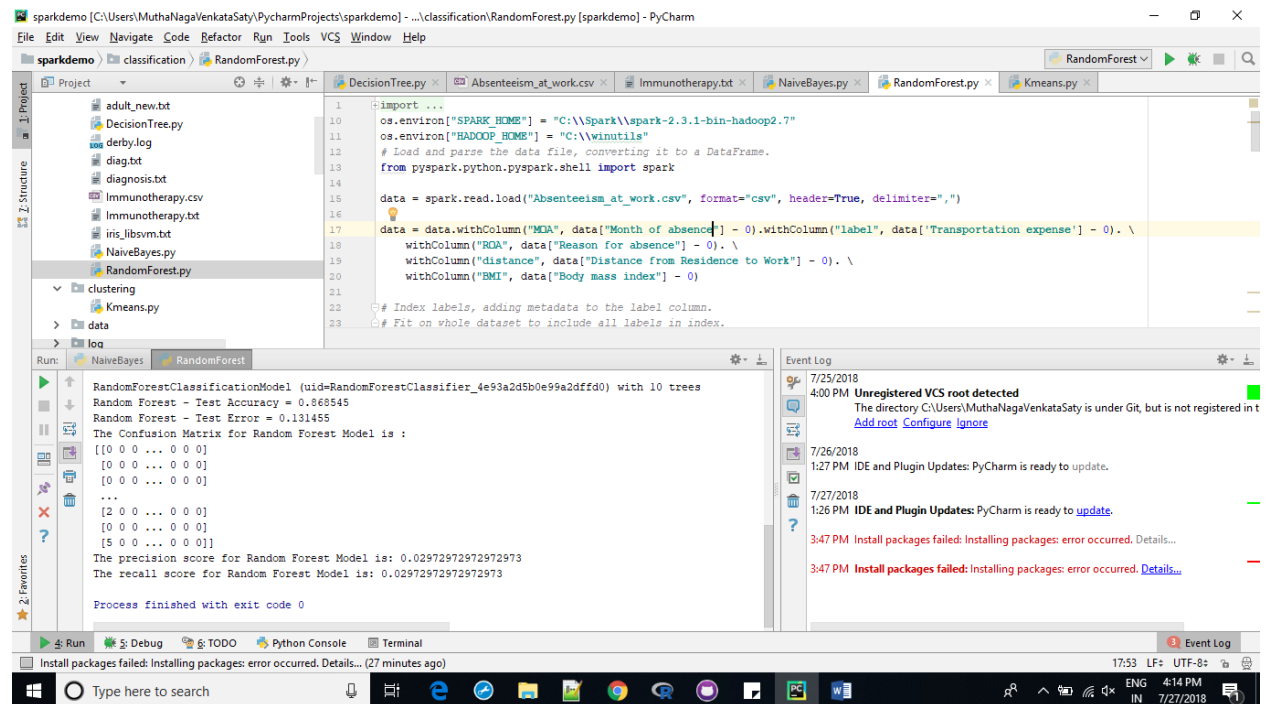
17 data = data.withColumn("MOA", data["Month of absence"] - 0).withColumn("label", data["Transportation expense"] - 0). \
18   withColumn("ROA", data["Reason for absence"] - 0). \
19   withColumn("distance", data["Distance from Residence to Work"] - 0). \
20   withColumn("BMI", data["Body mass index"] - 0)
21 # Index labels, adding metadata to the label column.
22 # Fit on whole dataset to include all labels in index.
23 assem = VectorAssembler(inputCols=["label", "distance"], outputCol='features')
24 data = assem.transform(data)
25 labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
26 # Automatically identify categorical features, and index them.
27 # Set maxCategories so features with > 4 distinct values are treated as continuous.
28 featureIndexer = \
29   VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)
30 # Split the data into training and test sets (30% held out for testing)
31 (trainingData, testData) = data.randomSplit([0.7, 0.3])
32 # Train a RandomForest model.
33 rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)
34 # Convert indexed labels back to original labels.
35 labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
36   labels=labelIndexer.labels)
37 y_true = data.select("BMI").rdd.flatMap(lambda x: x).collect()
38 y_pred = data.select("ROA").rdd.flatMap(lambda x: x).collect()
39 # Chain indexers and forest in a Pipeline
40 pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])
41 # Train model. This also runs the indexers.
42 model = pipeline.fit(trainingData)
43 # Make predictions.
44 predictions = model.transform(testData)
45 # Select example rows to display.
46 predictions.select("predictedLabel", "label", "features").show(5)

```

Random Forest Confusion Matrix:



Random Forest Results:



```
1 import ...
2 os.environ["SPARK_HOME"] = "C:\\Spark\\spark-2.3.1-bin-hadoop2.7"
3 os.environ["HADOOP_HOME"] = "C:\\winutils"
4 # Load and parse the data file, converting it to a DataFrame.
5 from pyspark.python.pyspark.shell import spark
6
7 data = spark.read.load("Absenteeism_at_work.csv", format="csv", header=True, delimiter=",")
8
9 data = data.withColumn("MOA", data["Month of absence"] - 0).withColumn("label", data["Transportation expense"] - 0). \
10 withColumn("ROA", data["Reason for absence"] - 0). \
11 withColumn("distance", data["Distance from Residence to Work"] - 0). \
12 withColumn("BMI", data["Body mass index"] - 0)
13
14 # Index labels, adding metadata to the label column.
15 # Fit on whole dataset to include all labels in index.
```

Run: NaiveBayes RandomForest

RandomForestClassificationModel (uid=RandomForestClassifier_4e93a2d5b0e99a2dffd0) with 10 trees
Random Forest - Test Accuracy = 0.868545
Random Forest - Test Error = 0.131455
The Confusion Matrix for Random Forest Model is :
[[0 0 ... 0 0]
[0 0 ... 0 0]
[0 0 ... 0 0]
...
[2 0 ... 0 0]
[0 0 ... 0 0]
[5 0 ... 0 0]]
The precision score for Random Forest Model is: 0.02972972972973
The recall score for Random Forest Model is: 0.02972972972973
Process finished with exit code 0

Event Log

- 7/25/2018 4:00 PM Unregistered VCS root detected
The directory C:\Users\MuthaNagaVenkataSaty is under Git, but is not registered in t
[Add root](#) [Configure](#) [Ignore](#)
- 7/26/2018 1:27 PM IDE and Plugin Updates: PyCharm is ready to update.
- 7/27/2018 1:26 PM IDE and Plugin Updates: PyCharm is ready to [update](#).
- 3:47 PM Install packages failed: Installing packages: error occurred. Details...
- 3:47 PM Install packages failed: Installing packages: error occurred. [Details...](#)

Conclusion:

	Decision Tree	Naïve Bayes	Random Forest
Accuracy (%)	96.7	15.45	86.85
Precision	0.0297	0.0297	0.0297
Recall	0.0297	0.0297	0.0297
Error (%)	3.2	8.6	13.14

From the above three results, we achieved highest accuracy for Decision Tree. It also has the least error rate around 5%.

This was a bit unexpected for me. Since the Random Forests are an ensemble of decision trees, it should give the optimum accuracy and output values. But surprisingly, Decision Tree has more accuracy than the Random Forests. This may be due to the percentage error which is the least for Decision Tree.

Part - 2

Objective:

The objective of this lab was to use Spark Streaming to track word usage across twitter. Spark Streaming gives us the ability to process data in real-time, as it is coming in (on selected time intervals). This can be very useful for things such as stock market analysis, or customer sentiment.

Data Description:

The data consists of all tweets submitted to twitter that occur while the program is running. Unfortunately, in order to access twitters API and get access to the twitter stream, you need a developer account. I created a new twitter, and applied for a developer account, but the delay between application and approval can take a while. Without API access, it is impossible to access the twitter stream, making the actual running of this program impossible.

Code:

```
def sendData(c_socket):
    #set authentication keys
    auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
    auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)

    #create connection
    twitter_stream = Stream(auth, TweetsListener(c_socket))

    #filter for tweets using term Trump
    twitter_stream.filter(track=['collusion'])

if __name__ == "__main__":
    #create and bind a socket to allow connection
    s = socket.socket()
    host = "localhost"
    port = 8888
    s.bind((host, port))

    #print("Listening on port: %s" % str(port))

    s.listen(5) # Now wait for client connection.
    c = s.accept()
    addr = c # Establish connection with client.

    #print("Received request from: " + str(addr))

    sendData(c)
```

Streaming:

```
7 from pyspark import SparkContext
8 from pyspark.streaming import StreamingContext
9
10 """
11 This is use for create streaming of text from txt files that creating dynamically
12 from files.py code. This spark streaming will execute in each 3 seconds and It'll
13 show number of words count from each files dynamically
14 """
15
16
17 def main():
18     sc = SparkContext(appName="PysparkStreaming")
19     ssc = StreamingContext(sc, 3) #Streaming will execute in each 3 seconds
20     lines = ssc.textFileStream('log') #'log/ mean directory name
21     counts = lines.flatMap(lambda line: line.split(" ")) \
22         .map(lambda x: (x, 1)) \
23         .reduceByKey(lambda a, b: a + b)
24     counts.pprint()
25     ssc.start()
26     ssc.awaitTermination()
27
28
29 if __name__ == "__main__":
30     main()
```

Word Count:

```
10 def convertTweet(tweet):
11     #changes json to string and splits at spaces
12     tweetStr = tweet.encode('utf-8')
13     tweetDict = json.loads(tweetStr)
14     text = tweetDict["text"]
15     splitTweet = (text.encode('utf-8')).split(' ')
16     return splitTweet
17
18 spark = SparkContext("local", "twitter wordCount")
19 sparkStream = StreamingContext(spark, 10)
20 IP = "localhost"
21 Port = 8888
22 lines = sparkStream.socketTextStream(IP, Port)
23
24 #convert/split each tweet into a flatmap
25 splitWords = lines.flatMap(lambda line: convertTweet(line))
26
27 #map each word to a value of 1, then reduce by key
28 wordCount = splitWords.map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y)
29
30 #print results
31 wordCount.pprint()
32
33 sparkStream.start()
34 sparkStream.awaitTermination()
```

Evaluation:

The actual word Count part of the program was dead simple. We first have to pull in the tweets as Json objects. Then we convert them to strings, split along the white space. We can flatmap these split strings and create a 'list' of all words used. Then we can do a very simple map reduce, where we map each word to the value of 1 originally, then reduce them by key to combine the values of all same words. The results are then printed to console.

Conclusion:

The actual mapReduce part of this assignment was quite simple at this point. But the connecting to twitter was somewhat more involved. I was able to find tons of good resources to help, the main one being <http://adilmoujahid.com/posts/2014/07/twitter-analytics/> . However, what I was completely unable to find was a way to connect to twitters API without first having access_keys and consumer_tokens. From the looks of older results, it used to be much easier to access the API, however with the rise of BOTS and the twitter events surrounding the 2016 election, I can understand twitter controlling api access much more closely now.